Application-Bypass Broadcast in MPICH over GM[*]

Darius Buntinas[‡] Dhabaleswar K. Panda[‡] Ron Brightwell[‡‡]

[‡]Network-Based Computing Laboratory
Dept. of Computer and Information Science
The Ohio State University
{buntinas, panda}@cis.ohio-state.edu

[‡‡]Scalable Computing Systems Dept.
Sandia National Laboratories[†]
bright@cs.sandia.gov

Abstract

Processes of a parallel program can become unsynchronized, or skewed, during the course of running an application. Processes can become skewed as a result of unbalanced or asymmetric code, or through the use of heterogeneous systems, where nodes in the system have different performance characteristics, as well as random, unpredictable effects such as the processes not being started at exactly the same time, or processors receiving interrupts during computation. Geographically distributed systems may have more severe skew because of variable communication times. Such skew can have a significant impact on the performance of collective communication operations which impose an implicit synchronization. The broadcast operation in MPICH is one such operation. An application-bypass broadcast operation is one which does not depend on the application running at a process to make progress. Such an operation would not be as sensitive to process skew. This paper describes the design and implementation of an application-bypass broadcast operation. We evaluated the implementation and find a factor of improvement of up to 16 for application-bypass broadcast compared to non-application-bypass broadcast when processes are skewed. Furthermore we see that as the system size increases, the effects of skew on non-application-bypass broadcast also increase. The application-bypass broadcast is much less sensitive to process skew which makes it more scalable than the non-application-bypass broadcast operation.

1. Introduction

Process skew is an important aspect in parallel and distributed systems which has not received much attention. Many collective communication benchmarks [12, 14] perform the collective communication with all processes starting the operation at the same time. While this would be ideal when running a parallel application, it is not realistic. Processes can become skewed as a result of unbalanced code, where one process has more computation to perform than others, of asymmetric code, where different processes perform different computation or communication operations, of using heterogeneous systems, where nodes in the system have different performance characteristics, as well as of random, unpredictable effects such as the processes not being started at exactly the same time, or processors receiving interrupts during computation. Process skew may be more severe in georgaphically distributed computing systems, where communication time between remote nodes may be variable. The effects of process skew become more severe as the size of the system grows.

Collective communication operations can impose implicit synchronization. When processes are skewed, such synchronization will cause certain processes to wait idle for other processes to catch-up. With certain collective communication operations this synchronization is unavoidable unless a split-phase approach is used, such as a *reduce-to-all* operation where all processes must provide input and start the operation before any can finish, and with others, such as *barrier*, synchronization is the desired effect. However for other collective operations such as *broadcast* and *reduce-to-one* it is desirable to reduce the amount of implicit synchronization in order to reduce the effects of skew and improve overall system performance.

For instance, the broadcast operation in MPICH [8] is implemented such that a process will not forward the broadcast message until that process has made a call to `MPI_Bcast()` and received the message. If a process is slow to call `MPI_Bcast()`, other processes may be delayed as a result. A more desirable implementation would be to allow the broadcast operation to *bypass the application*. The concept of application-bypass operations was discussed in [3]. We will describe application-bypass in detail in the next section.

[*]This research is supported in part by a DOE grant #DE-F002-01ER25506 and NSF grants #EIA-9986052 and #CCR-0204429.

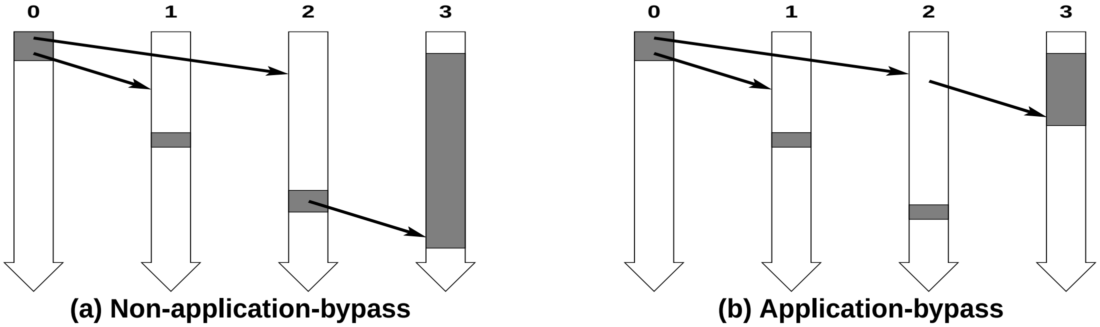


Figure 1. Broadcast operation over four processes. The large arrows represent timelines for each process. The shaded areas in these timelines represent a call by the application to the broadcast function, and the small arrows represent broadcast messages.

We have designed and implemented an application-bypass broadcast operation by modifying GM [11] and MPICH over GM. We then evaluated our implementation and found that using application-bypass broadcast we see a factor of improvement of up to 16 when the processes are skewed. We also noticed that the effects of process skew become more severe as system size increases, and that application-bypass broadcast is considerably less sensitive to process skew than non-application-bypass broadcast. This indicates that that an application-bypass approach is critical for dealing with process skew allowing collective communications operations to be scalable.

The outline of the rest of the paper is as follows. In the next section we describe the basic idea of application-bypass. The design and implementation of our application-bypass broadcast operation are described in Section 3. We evaluate our implementation in Section 4, the conclude in Section 5.

2. Application-bypass

The basic idea of an application-bypass operation is that the application need not be involved in order for the operation to proceed. Broadcast is an operation which can be implemented in an application-bypass manner. The broadcast operation in many message passing systems is performed by creating a logical binomial tree over the processes participating in the broadcast. The root process sends a copy of the data to each of its children. Each non-root process waits to receive the data, then sends copies of the data to each of its children, if any. At the application level, each process pariticipating in the broadcast will call the broadcast function.

In MPICH, the `MPI_Bcast()` function performs the broadcast by first waiting for the message to be received from the parent process, if this process is not the root, then sending copies of the message to each child process. The broadcast operation in MPICH is not implemented in an application-bypass manner. If a message is received by a process, the message will not be forwarded to its children until the process calls `MPI_Bcast()`. This means that if a non-root, non-leaf node is delayed, the descendants of that process will also be delayed, even if they have called `MPI_Bcast()`, and are waiting for the message. If the broadcast operation were implemented in an application-bypass manner, as soon as a process receives a broadcast message, it would forward the message to its children, regardless of whether the process has called `MPI_Bcast()`.

Figure 1 illustrates the concept of an application-bypass broadcast. Figure 1(a) shows a non-application-bypass broadcast, while Figure 1(b) shows an application-bypass broadcast. These diagrams show four processes, the root process, Process 0, sends messages to its children, Processes 2 and 1. Process 2 receives the message and sends it to Process 3. Notice that in this example, the processes do not call broadcast at the same time, in particular, Process 2 calls the broadcast call much later than Processes 0 and 3. Because of this, in the non-application-bypass case show in Figure 1(a) we see that even though Process 3 had called the broadcast function and was waiting for the message, it did not receive the message until after Process 2 finished its computation and called the broadcast function.

In the application-bypass case shown in Figure 1(b), as soon as the broadcast message arrives at Process 2, it receives the data into a temporary buffer, and sends a copy to Process 3. Process 3 can receive the message much sooner because it doesn't have to wait for Process 2 to call the broadcast function. Once Process 2 calls the broadcast function, it will copy the data for the broadcast message from the temporary buffer to its final location.

Application-bypass operations can be even more important in large scale or heterogeneous systems. In such systems it is more likely for processes to be skewed, and so collective communication operations may not be called at the same time by all of the processes. Non-application-bypass operations can impose implicit synchronization among the processes, which means some faster processes will sit idle waiting for slower processes to catch up. Application-bypass operations can reduce the amount of synchronization that such operations cause. This can reduce the amount of time processes spend waiting for each other and can improve overall application performance.

We note that some networks provide a broadcast primitive. Such broadcast operations bypass the application because they are performed by the network, and not the host. In this paper, however, we describe application-bypass broadcast on a netowrk which does not provide the broadcast primitive.

3. Design and implementation

In this section we will describe the design and implementation of application-bypass broadcast. We start by identifying some design alternatives which we considered, next we give an overview of MPICH over GM, then describe our implementation in detail.

3.1. Design alternatives

We identified several options for implementing application-bypass broadcast. One design option would be to use a *broadcast thread* to perform the broadcast operation. To perform a broadcast, the main thread would send a message to its broadcast thread. The broadcast thread would be polling for incoming messages and would broadcast the message among the broadcast threads associated with the other processes. After broadcasting the message, the broadcast threads would send the message to their main thread. Because the broadcast thread is constantly polling for incoming messages it consumes processor resources which could be better used by its main thread, on a uniprocessor system, or by additional computation threads on an SMP system. For this reason this option may not be practical in a real system.

Another alternative would be to have the broadcast thread block while waiting for an incoming message. This option would not waste processor resources, but would increase the latency of performing a broadcast because of the interrupt overhead. The cost of performing interrupts for every broadcast may make this option impractical.

We chose to implement a third option which uses a single thread and a signal handler. This option does not waste processor resources because the signal hander is only called when a message needs to be processed. Furthermore, because there is only one thread, when the thread is polling for a message, there is no need for a signal to be generated to process an incoming message. Our implementation allows the thread to disable interrupts at the NIC when polling for a message. This gives us the best of both previous design alternatives: low latency broadcast and low processor usage overhead.

3.2. Overview of GM and MPICH over GM

Before we describe our implementation we will briefly describe some internal details of GM and MPICH over GM. Figure 2 shows the software and hardware layers associated with MPICH over GM. GM is a user-level communication subsystem over the Myrinet [2] network. The Myrinet network is a 2Gbps full duplex network with network interface cards (NICs) that have programmable processors. GM consists of a kernel component, a user-level library and a NIC component. The kernel component is used for things like setting up new communication endpoints, and registering memory, and is not used in the critical path. The NIC component is code which is executed on the NIC processor. Almost all of the protocol processing is performed at the NIC. The user-level library is basically used as an interface between the host process and the NIC code.

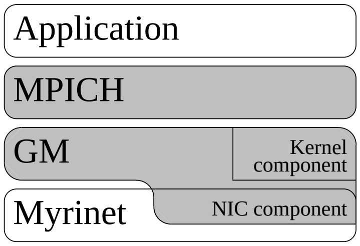


Figure 2. Software and hardware layers for MPICH over GM

MPICH is an implementation of the MPI [10] message passing interface standard. MPICH has been ported to many different platforms and networks including GM.

The broadcast operation in MPICH is performed, as described earlier, by propagating messages over a broadcast tree. Each process participating in the broadcast makes a call to `MPI_Bcast()`. The root node and source or destination buffer are specified as parameters to this function. A *communicator* is also given as a parameter which specifies the group of processes which will participate in the broadcast. In the call to `MPI_Bcast()`, each process determines its parent process, if any, then waits to receive the message from this process. Once the message is received, it determines which processes are its children and sends the message to them.

When a process makes a call to receive a message, MPICH checks to see if the message has already been received. It searches a queue called the *unexpected message queue* for messages which match certain criteria specified in the receive call, such as sender id, datatype, and tag. If the message is not found, a *descriptor* is posted describing the anticipated message, as well as the memory location where the data should be received into. MPICH will then optionally poll for incoming messages until the message has been received.

When a message arrives at a process, MPICH first checks the list of posted receives, to see if this message is expected. If it is expected, it copies the data to the location specified in the descriptor, then marks the descriptor that the receive has completed. If MPICH finds no posted descriptor matching the incoming message, the message is copied into the unexpected queue.

MPICH over GM uses two modes in sending messages: *eager* and *rendez-vous*. The eager mode is used to send small messages. In this mode the data for the message is copied into a send buffer and is transmitted from the buffer. The copy to a send buffer is necessary because GM can only send data which is located in a pinned memory region. Pinning a memory region requires a system call and so is faster to copy the data to a pre-pinned buffer and send it from there than to perform the system call to pin the data in its original location. When the message is received, GM places the message in a pre-pinned buffer at the receiver. The data for the message must then be copied out of this buffer to its final location.

For large messages because the cost of copying the data becomes quite large, it is faster to pin the memory of the original source of the data at the sender and the final destination at the receiver, then send the data directly from the original location to the final destination eliminating any copies of the data. The rendez-vous protocol is used to perform this. The sender sends a request-to-send message to the receiver, pins the memory for the source of the data and waits for a response from the receiver. Upon receiving the request-to-send message, the receiver pins the memory for the final destination of the data and sends the address of this to the sender in a OK-to-send message. When the sender receives this message, it sends the data directly from the source location to the remote destination.

3.3. Our implementation

We modified MPICH over GM version 1.2.4..8a to provide application-bypass broadcast functionality. We also modified GM version 1.5.2.1 to allow signals to be generated when messages are received.

In MPICH, the broadcast operation is performed by a process when the application calls `MPI_Bcast()`. In order for the broadcast operation to bypass the application, the broadcast operation would have to be performed as soon as the broadcast message is received by the MPICH library. We did this by defining a new message type. When such a message type is received by the MPICH *progress engine*, copies of the message are sent to each of the children. Once the copies of the message are sent, the progress engine handles the message the same way as any other received message.

The list of children of a process is calculated by knowing the processes that are participating in the broadcast and which process is the root of the broadcast. Normally these parameters are supplied by the application to the `MPI_Bcast()` call. However, in our implementation, for non-root nodes, the broadcast operation is not performed in the `MPI_Bcast()` function. Instead, we added a field to the header of broadcast messages to identify the root. Also, MPICH includes a *context id* field in each message which can be used to uniquely identify an MPI *communicator*. A communicator specifies which processes are participating in a collective communication. When a communicator is created, we computed the list of children for that communicator, for each possible root. We store this array of lists of children in a hash table hashed on the communicator's context id. Then when an incoming broadcast message is received, we can get the list of children by getting the array from the hash table using the context id of the message, and indexing on the root, which is also given in the message.

In our implementation of application-bypass broadcast, we only considered messages sent in the eager mode. For MPICH over GM these are messages which are less than 16KB. With rendez-vous messages, the final destination must be known in order for the message to be sent. However, the final destination of the broadcast message is not known until the application calls `MPI_Bcast()`. Using a temporary buffer to store a broadcast rendez-vous message would require memory copies which would defeat the purpose of using the rendez-vous mode for large messages. We intend to study this issue in the future.

We added a signal handler which calls the progress engine to process any new messages. In order to avoid race conditions we added a mutex variable which is set when the process calls the progress engine. When the signal handler is called, it checks the mutex variable and exits if it is set. The mutex variable is reset when the process exits the progress engine. However, this could lead to a case where we could lose a signal for a new broadcast message. For example, a new broadcast message could be received just before the process left the progress engine. The signal handler would be called, but it would exit immediately, because the mutex variable indicates that the process is executing the progress engine. When the process continues executing, it would leave the progress engine without handling the newly received broadcast message. To deal with this situation, we added a loop around the progress engine which keeps calling the progress engine while there are messages waiting to be received. This way, any broadcast message received after the progress engine processed the last message, and before it resets the mutex will be handled, because the loop condition will find that there is a pending receive. Any broadcast message received after the mutex variable is reset will be handled by the signal handler.

GM does not have the capability to generate signals when a message is received. We modified GM to add this capability. Since performing an interrupt for every incoming message would have a severe impact on performance, we wanted to perform interrupts only when necessary. We did this by defining a new packet type in GM. Only the reception of these packets generates a signal. This way the sender of a message can specify that a signal will be generated for the receiving process when the message is received.

We also allow a process to disable the signal generation at the user level. We added a flag to the data structure at the NIC which is mapped into the process' address space. The process simply has to write to the flag to enable or disable signal generation. Since this flag is located in NIC memory any accesses by the host will go over the PCI bus. This is considerably slower than accessing local memory, and may interfere with other PCI traffic. For this reason care must be taken when accessing this flag not to adversely impact system performance.

We used the signaling capability that we added to GM to generate signals for sending broadcast messages to non-leaf nodes. This limits the number of interrupts to only those cases where it could benefit. Furthermore, we disabled signaling at the NIC whenever the process calls `MPI_Bcast()` or whenever the process is waiting for a receive. This way, if the process is already polling for a message, there is no need to generate a signal to have the broadcast message processed. By eliminating as many interrupts as possible, we reduce the impact of using a signal handler while giving us the advantages of application-bypass broadcast.

4. Experimental results

We performed our evaluation on a 32 node cluster consisting of 16 700MHz quad-SMP Pentium III nodes with 66MHz/64 bit PCI slots, and 16 1GHz dual-SMP Pentium III nodes with 33MHz/32 bit PCI slots. The cluster was connected using a Myrinet-2000 network. The network consists of 28 PCI64B cards with 133MHz LANai 9.1 processors and 4 PCI64C cards with 200MHz LANai 9.2 processors and are connected using fiber cables to a 32 port switch. Each of the nodes ran the 2.4.18 Linux kernel. Our tests were performed using GM version 1.2.5.1 and MPICH version 1.2.4..8a which are the same versions as our modified GM and MPICH.

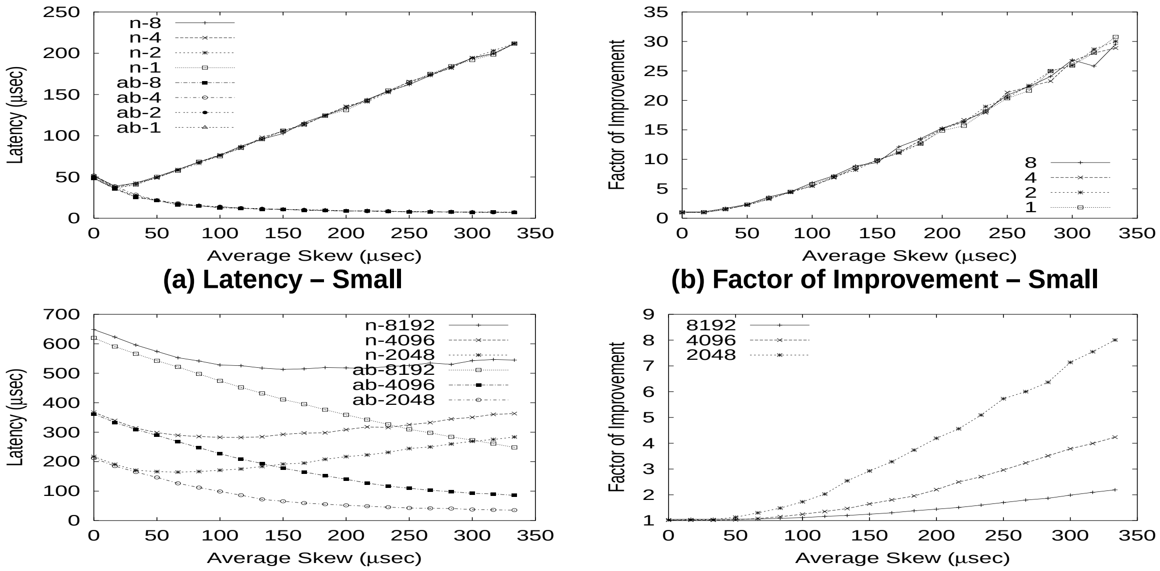
We evaluated our implementation using micro-benchmarks. The first micro-benchmark compares the average time to perform `MPI_Bcast()`. In this micro-benchmark, the root process calls `MPI_Bcast()` while the other processes perform a delay loop, followed by a call to `MPI_Bcast()`. This test is performed 1,000 times with an `MPI_Barrier()` being called before each test. The number of iterations a process performs in the delay loop is chosen randomly, between 0 and a maximum delay value, by each process each time the test is performed. Notice that increasing the maximum delay value, increases the skew between the processes.

Since we used a heterogeneous system, we wanted to normalize the delay loops. We did this by having each process count how many iterations it can compute in 50μs . The maximum delay value was then incremented by that many iterations. For the 700 MHz machines, this was about 17,500 iterations, while on the 1GHz machines this was about 25,000. The graphs show the *average* skew time in microseconds for convenience.

Figure 3 shows the average time each process spent in the `MPI_Bcast()` call over 32 processes. Figure 3(a) shows these results for 1, 2, 4 and 8 byte messages. The graph does not show much differentiation between the message sizes, however a large difference is seen between the application-bypass MPICH and non-application-bypass MPICH. Notice that as the skew increases, the average time spent in `MPI_Bcast()` by the non-application-bypass MPICH increases when the average skew is larger then 17μs . This is as we expected because as the skew between processes increases more processes are being delayed longer waiting for one of their ancestors to call `MPI_Bcast()`. However, for application-bypass MPICH, we see that the average time spent in `MPI_Bcast()` actually decreases as the skew increases. We don't see an increase as the skew increases as we did with the non-application-bypass MPICH because even if a non-leaf process is performing the delay loop when a broadcast message is received, the process will be interrupted and the broadcast operation will be allowed to proceed. The reason why the average time spent in `MPI_Bcast()` actually decreases is because as the time each process spends in the delay loop increases, the probability that the broadcast message has arrived and that the broadcast operation has completed before the process calls `MPI_Bcast()` increases. If the broadcast message has arrived before the process calls `MPI_Bcast()`, then all that needs to be performed in the `MPI_Bcast()` function is to copy the received data to the final memory location. Notice that when the average skew is 17μs , the time the non-application-bypass MPICH spends in `MPI_Bcast()` decreases compared to when there is no skew. This is because some of the time that processes lower down in the broadcast tree would spend waiting for the broadcast message to propagate down the tree is overlapped with the delay loop. Although the broadcast message may be delayed because a process higher up in the tree is delayed, this delay is smaller than the time which is overlapped.

Figure 3(b) shows the factor of improvement of performing broadcasts using application-bypass MPICH over non-application-bypass MPICH for small messages. The graph shows a factor of improvement of up to 30 for the time spent in `MPI_Bcast()` when the average skew is 333μs and when broadcasting a 1 byte message. The improvement for 2, 4 and 8 byte messages is similar.

We performed the same evaluations using large messages. Figure 3(c) shows the results for 2K, 4K and 8K messages. Again we see that the time spent



(a) Latency – Small

(b) Factor of Improvement – Small

(c) Latency – Large

(d) Factor of Improvement – Large

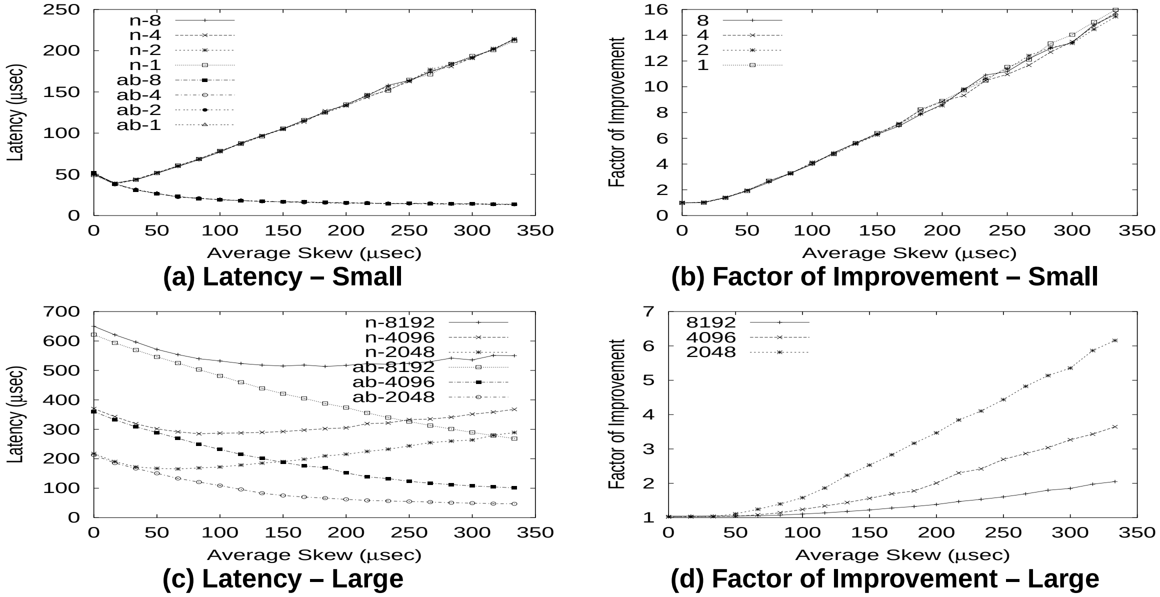
Figure 3. Average latency of MPI_Bcast function on 32 nodes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab)

in `MPI_Bcast()` for application-bypass MPICH processes decreases as the skew increases, while for non-application-bypass MPICH processes the time increases once the skew is large enough that the effect of the overlap of the broadcast and delay loop is no longer seen. Notice that even when the skew is small or even zero, the application-bypass MPICH processes spend less time in `MPI_Bcast()` than the non-application-bypass MPICH processes. We believe that this is because even when the skew is zero, and the processes spend no time in the delay loop, the processes are still skewed slightly due to the nature of a distributed system. Even though a `MPI_Barrier()` is called before each test, not all processes will leave the barrier at exactly the same time. It is possible that some processes may receive the broadcast message while still performing the barrier. In application-bypass MPICH, when the message is to be forwarded to the child processes, the data will be copied from the receive buffer into the send buffers and the messages are sent to the children. In non-application-bypass MPICH, the message is copied into the unexpected queue. When `MPI_Bcast()` is called, the data is copied out of the unexpected queue and into the final memory location, then the data is copied from this memory location and into the send buffers to be sent to the child processes. Notice that the non-application-bypass MPICH has an extra memory copy in the critical path, which explains why we see a larger difference for larger message sizes when the skew is small.

Figure 3(d) shows a factor of improvement in the time spent in the `MPI_Bcast()` function of up to 8 for 2K messages, up to 4.2 for 4K messages, and 2.2 for 8K messages when the average skew is 333µs .

Just considering the time a process spends in `MPI_Bcast()` does not consider the time the application-bypass MPICH spends performing the operation when a broadcast message is received before the call to `MPI_Bcast()` is made. In order to evaluate the impact of performing the broadcast operation asynchronously and of the associated interrupts on the computation, we first timed the delay loop when there are no incoming broadcast messages. Then we timed the total time the process spends in the delay loop and the `MPI_Bcast()` call, and subtracted off the time it would have spent in the delay loop had there been no incoming broadcast messages. What is left is the time the process spends broadcasting. Figure 4 shows the results of these tests for 32 processes. Again the graph for small messages, Figure 4(a), does not show much difference between the different message sizes, but a significant difference is seen between the application-bypass MPICH and non-application-bypass MPICH. Notice that these results are very similar to those for just the time spent in `MPI_Bcast()`. There is about a 6µs overhead in the application-bypass case for processing broadcast messages by the signal handler for small messages. Figure 4(b) shows a factor of improvement of up to 16 for application-bypass MPICH processes over non-application-bypass MPICH processes. This is a significant improvement over non-application-bypass MPICH.

Figure 4(c) shows the results of the same test for large messages. As with the small messages, the results are



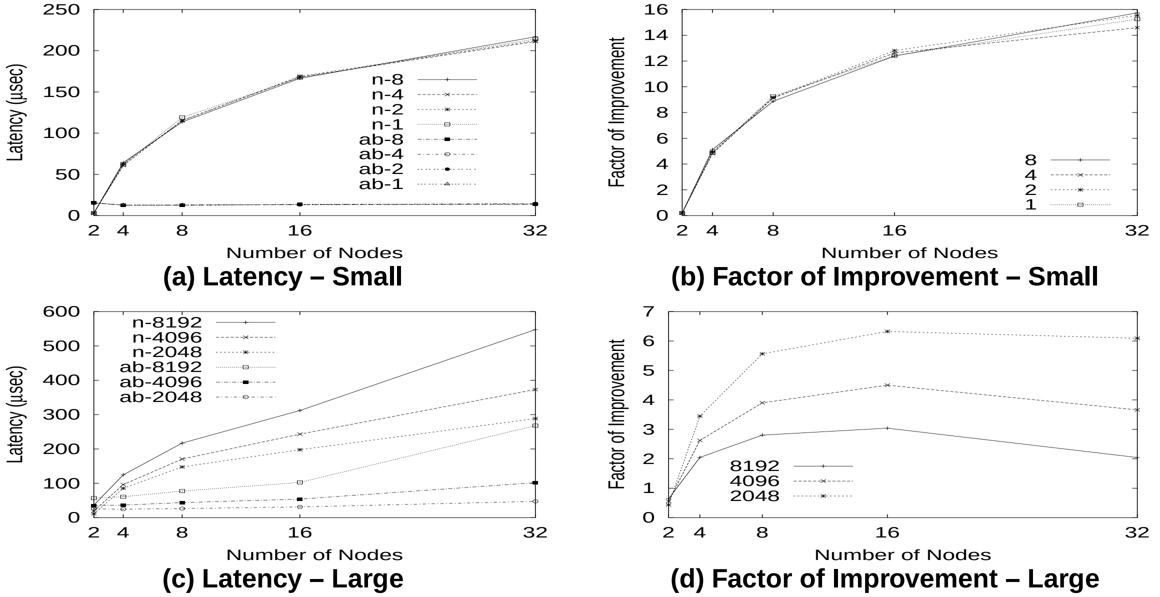
(a) Latency – Small

(b) Factor of Improvement – Small

(c) Latency – Large

(d) Factor of Improvement – Large

Figure 4. Average latency of signal handler and MPI_Bcast function on 32 nodes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab)



(a) Latency – Small

(b) Factor of Improvement – Small

(c) Latency – Large

(d) Factor of Improvement – Large

Figure 5. Average latency of signal handler and MPI_Bcast function with a average skew of 333µs for various number of processes. Small messages sizes are 1, 2, 4, and 8 bytes, and large message sizes are 2048, 4096, and 8192 bytes for non-application-bypass MPICH (n) and application-bypass MPICH (ab)

very similar to the results for just the `MPI_Bcast()` function. For large messages there is about a 11µs , 15µs , and 20µs overhead for processing 2K, 4K, and 8K broadcast messages in the signal handler, respectively. Figure 4(d) shows factors of improvement of up to 2 for 8K messages, 3.6 for 4K messages, and 6.2 for 2K messages. Again, these are significant improvements.

In order to see how application-bypass broadcast can benefit systems of different sizes, we performed a test similar to the previous, except we used an average skew of 333µs and varied the number of processes. As before, for 32 processes, we used both the 700MHz machines and 1GHz machines, but for 16 and fewer processes, only the 700MHz machines were used. Figure 5 shows these results. Notice that for small messages, in Figure 5(a), as the system size increases the time taken by the application-bypass MPICH processes remains almost constant. This is because for all but the two process case, all of the delay of propagating the message down the broadcast tree is overlapped by the delay loop. The increase in time seen in the non-application-bypass MPICH results as the number of nodes increases is due primarily to process skew. As the number of processes participating in the broadcast increases, the number of processes waiting in `MPI_Bcast()` for an ancestor to finish the delay loop and perform the broadcast also increases. We see a similar effect in Figure 5(c) for large messages. The time increases much slower as the system size increases for application-bypass MPICH versus non-application-bypass MPICH. These results indicate that the effects of process skew become more severe as system size increases. Furthermore, they indicate that an application-bypass approach is critical for dealing with process skew allowing collective communications operations to be scalable.

5. Conclusions and future work

We have described our design implementation of application-bypass broadcast and evaluated our implementation. Our evaluation shows that an application-bypass broadcast is not as sensitive to process skew as non-application-bypass broadcast. In fact, using the application-bypass broadcast, we have seen a factor of improvement of up to 16 when processes are skewed. Furthermore we see that in a non-application-bypass broadcast the effects of process skew increase as the system size increases. We note that while process skew can be reduced by careful design of parallel programs and close control of the computing environment, we believe that process skew cannot be eliminated altogether. For this reason, we believe that the use of application-bypass is critical to improving the scalability of collective communication operations, and of the system in general.

We intend to implement other collective communication operations in an application-bypass manner, such as reduction, scatter, and gather. We also intend to examine the possibility of implementing these operations as NIC-based operations. NIC-based operations are operations which are performed by the NIC processor rather than by the host [4, 6, 7, 5, 1, 13, 9, 15]. Because they are performed at the NIC and not the host these operations naturally bypass the application.

We also note that our implementation required modifying the NIC firmware to allow sending interrupts for certain messages. We feel that this is a worthwhile feature and should be provided as a standard feature to communication subsystems and NICs. However, we also intend to investigate how best to provide application-bypass features using networks which do not provide the selective interrupt feature.

Additional Information Additional papers related to this research can be obtained from the following Web pages: Network-Based Computing Laboratory (http://nowlab.cis.ohio-state.edu) and Parallel Architecture and Communication Group (http://www.cis.ohio-state.edu/∼panda/pac.html).

References

[1] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proc. of the 27th Int'l Conf. on Parallel Processing (ICPP '98)*, pages 381–390, August 1998.

[2] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[3] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Proc. of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[4] D. Buntinas, D. Panda, and W. Gropp. NIC-based atomic remote memory operations in Myrinet/GM. In *Workshop on Novel Uses of System Area Networks (SAN-1)*, February 2002.

[5] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages. In *Proc. of Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC)*, pages 115–129, 2000.

[6] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proc. of the Int'l Parallel and Distributed Processing Symposium 2001, (IPDPS)*, April 2001.

[7] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance benefits of NIC-based barrier on Myrinet/GM. In *Proc. of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[9] R. Kesavan and D. K. Panda. Optimal Multicast with Packetization and Network Interface Support. In *Proc. of Int'l Conf. on Parallel Processing*, pages 370–377, Aug 1997.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[11] Myricom. Myricom GM myrinet software and documentation. http://www.myri.com/scs/GM/doc/gm_toc.html.

[12] Pallas MPI benchmarks - PMB, part MPI-1. ftp://ftp.pallas.com/pub/PALLAS/PMB/PMB-MPI1.pdf.

[13] R. Sivaram, R. Kesavan, D. K. Panda, and C. B. Stunkel. Where to Provide Support for Efficient Multicasting in Irregular Networks: Network Interface or Switch? In *Proc. of the 27th Int'l Conf. on Parallel Processing (ICPP '98)*, pages 452–459, August 1998.

[14] Sphinx parallel microbenchmark suite. http://www.llnl.gov/CASC/sphinx/sphinx.html.

[15] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proc. of the Int'l Conf. on Parallel Processing*, pages III:156–165, Aug 1996.